Available online at www.sciencedirect.com

SciVerse ScienceDirect

Procedia Computer Science 18 (2013) 601 – 610

Procedia Computer Science

International Conference on Computational Science, ICCS 2013

CarSh: A Commandline Execution Support for Stream-based Acceleration Environment

Shinichi Yamagiwa[a,c,*], Shixun Zhang[b]

[a] *Faculty of Engineering, Information and Systems,University of Tsukuba*
[b] *Department of Computer Science,University of Tsukuba*
1-1-1 Tennodai, Tsukuba, Ibaraki, 305-8573, Japan
[c] *JST PRESTO*

Abstract

The stream computing using manycore architecture such as GPU and the accelerators on FPGA has become one of the main methods for achieving high performance computing that such accelerators are employed in the recent top supercomputers. The stream computing implements an implicit concurrent program execution in massively parallel architecture applying, for example, OpenCL runtime. Although the potential high performance is achieved by the accelerator, programmers need to consider two kinds of programs; one is the control program on the host CPU such as buffer management for I/O data and invocation timings for the kernel program on the accelerator, and another is the kernel program itself executed by the accelerator. To eliminate this double programming difficulty, this paper proposes a new execution tool for the accelerator programs called CarSh providing a commandline-based interface that receives executable file described by XML over the flow-model framework of the Caravela platform. CarSh also provides the virtual buffer function to exchange the data streams from one kernel program to another. It eliminates explicit physical buffer management on the host CPU side from the programmer. Through the evaluations regarding performance and programmability, this paper concludes that CarSh implements a simple and transparent programming interface for the stream computing.

Keywords: Stream computing: Multicore system: Manycore system: Accelerator: GPU: Programming support

1. Introduction

A new technique for exploiting parallelism on an algorithm executed on a computing LSI chip has been promoted by the multicore/manycore technologies. In those environments, the multiple threads as many as the number of the computing cores are concurrently invoked in parallel and perform the entire computation. Two types of approaches are employed in the recent fashion: increasing the number of high performance CPU cores and increasing the one of small computing units. While the former one targets to exploit coarse-grain parallelism based on process or thread level, the latter one focuses on element level parallelism applying hundreds of small computing units. Therefore, the manycore architecture is recently employed as accelerators equipped as the neighborhood of the multicore CPUs to help a part of the entire computing. GPU (Graphics Processing Unit) is one of the successful examples of the manycore architectures that provide massively parallel environments where hundreds of small threads are invoked concurrently [1].

*Corresponding author.
E-mail address: yamagiwa@cs.tsukuba.ac.jp.

Selection and peer review under responsibility of the organizers of the 2013 International Conference on Computational Science
doi:10.1016/j.procs.2013.05.224

Although the accelerators such as GPUs provide powerful computing environments, those are coupled with CPU via the peripheral bus such as PCI Express in the system [2]. Therefore, the execution environments of the control and the computation programs must be separately written using the corresponding languages to the different execution environments [1]. In such manycore architecture, we focus on exploiting element level parallelism from entire algorithm. A typical computing style is called as the *stream computing* that lets the programmer consider that the output data forms a stream identified by the element index [3]. The stream computing leads us to program implicit parallelism in a program.

Standardized stream-based languages has been proposed and used in many situations where need massively parallel execution of applications. For example, CUDA [4] is a de fact standard language executed on GPU and provides the runtime API invoked on the CPU side for dedicated to NVIDIA GPUs. The OpenCL [5][6][7] is a standard programming environment that provides a seamless support for the accelerators for massively parallel environments. These languages makes to program on the accelerators easy for us, especially using the OpenCL, due to the portable interface for any types of the accelerators with massively parallel environment. However, any of the languages force us to write both the control and the computing programs in different worlds of the computing architecture and resources such as the embedded programing style. The control program must be invoked in the host environment such as the conventional CPU to configure the computing program to the accelerator, to download/upload the input/output data to/from the accelerator. The computing program (generally called *kernel* program) must be written in the stream-based style that fits to the architecture of the accelerator using the data communicated with the control program. Thus, this complexity for the double programming degrades the program productivity and easily produces bugs in the program. Therefore, it is indispensable to develop a revolutionary environment for programming the accelerator with focusing on just development of the kernel program.

This paper proposes a novel shell-like commandline-based environment called *CarSh* for executing the stream-based programs on the accelerator, where allows the programmer to execute the kernel program without making the control one. The CarSh implements the execution mechanism using the *flow-model* framework proposed in the Caravela environment [8][9]. The flow-model capsules the kernel program executed in the accelerator, I/O data streams and the parameters for parallelization in the accelerator packed into an XML file. Applying the CarSh, the programmer just needs to define a flow-model and then executes it from the commandline of the CarSh. This paper also proposes a new mechanism for pipeline execution availability using multiple flow-models on the CarSh, providing *virtual buffer*s that virtually implements temporal space for I/O data passed during pipeline execution of the flow-models.

The rest of this paper begins from the research backgrounds and mentions the recent environments for stream-based programs on the accelerators. Section 3 describes the design and implementation of the CarSh commandline shell for stream-based programs. Section 4 shows evaluation of the CarSh focusing on the aspects of performance and programmability. Finally, section 5 concludes the paper and shows future plans.

2. Research Backgrounds

2.1. Stream-based Computing on Accelerators

Due to the fast growing of market of multicore/manycore architecture, it is trend to apply a parallel programming approach exploiting thread level parallelism from applications, and to invoke it at the suitable architecture. The number of CPU cores is getting increased in an LSI chip, and thus the environment for invoking multiple threads/tasks becomes available in our desktop, laptop computers and also in mobile devices. Thus, the parallel and distributed computing technique on the *multicore* technology [10] has become available to meet anywhere in our life. However, because a CPU core is too large to integrate hundreds of it on an LSI chip, another approach to achieve highly parallel execution of many threads was invented and commercially provided in the market. This tide of new architecture is called *manycore*.

Recently, graphics processing demands a fast computation to achieve a high frequency framing on a dynamic graphics in especially entertainment market [2], structuring thousands of graphics objects textured with color tiles at every frame [11]. Such graphics processing is performed by small processing units that are programmable for each graphics processing during rendering an image frame [12]. Therefore the multiple pixels are concurrently computed in the hundreds of units. The programmer uses this potential high performance that is achieved by many

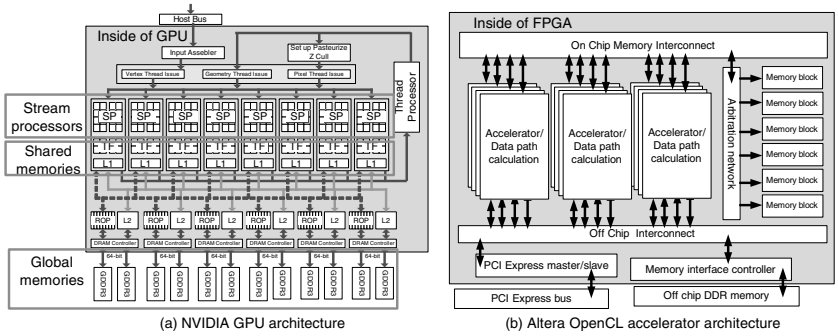


Fig. 1. Manycore architecture examples of NVIDIA GPU and Altera OpenCL Accelerator.

computing units concurrently working. It also exploits the fine-grained parallelism from application program implicitly [13].

Fig. 1 shows example architectures of manycore computing system. The manycore architecture implements massive small computing units in an LSI. In the GPU case as depicted in Fig. 1(a), the stream processors work as the computing units corresponding to individual calculations for the corresponding data elements given by the application. The number of stream processors is, for example, 448 in the NVIDIA Tesla GPU. As all stream processors are performing computations concurrently, it invokes massively parallel computation, thus achieves high computational performance. Another architecture illustrated in Fig. 1(b) is the Altera's accelerator for OepnCL framework [14]. It also implements from hundreds to thousands of reconfigurable hardware accelerators. The accelerators work concurrently and perform the computation for application as well as the GPU case.

The manycore architecture performs a different computation mechanism from the conventional CPU-based one. Each processing unit identifies its target computing element regarding a processor index. For example, assuming a vector summation $r_c = r_a + r_b$. The programmer needs to consider that the calculation is separated to each element of vectors like $r_c[id] = r_a[id] + r_b[id]$, where the id is the index of the vector element. Each calculation for $r_c[id]$ is assigned to a computing unit, then the summations of elements in the vector are performed in parallel. Optimistically, the vector summation needs only the processing time to calculate the "+" operation when the number of computing units is larger than the length of r_c. Thus, the programmer needs implicitly to consider the indexing of computing elements and also the independent computation for each computing element assigned to a unit. Here the left side of the computing equation (i.e. $r_c[id]$ above) is calculated like a data stream outputted in the ascending order identified by the increasing index. Therefore, it is called the *stream computing*.

The computing style of the manycore architecture does not fit to control operations for resources equipped inside/outside of the LSI chip due to the stream-based computation. Therefore, it works as an accelerator of the host CPU connected via the peripheral bus. Both architectures of Fig. 1 include the interfaces to the peripheral buses. The bus is used for sending/receiving the computing program and the input/output data for the calculation to/from the accelerator side. Therefore, for programming the manycore architecture, the host CPU is indispensable and controls the configuration and the behavior of the accelerator. Thus, the programmer must make both the computing program on the accelerator side and the controlling program on the host CPU side inevitably.

2.2. Stream-oriented Programming Environment

To reduce the difficulty of the double programming situation, there exist programming languages and the runtimes. The recent de fact standard ones are NVIDIA's CUDA [4] and OpenCL [5][6][7].

The CUDA assumes an architecture model as illustrated in Fig. 2 (a). The model defines a GPU which is connected to a CPU's peripheral bus. A VRAM maintains data used for calculation on the GPU. The kernel program is downloaded by the host CPU to GPU and the data is also copied from the host memory. The program is executed as a thread in a thread block grouped with multiple threads. The thread blocks are tiled in a matrix of from one to three dimensions. In the figure, thread blocks are tiled in two dimensions which size is $n_{grid} \times m_{grid}$. Each thread block consists of $n_{block} \times m_{block}$ threads. The program shown in Fig. 2 (b) is an example of vector summation written by using CUDA. The kernel program is defined with the `__global__` directive so that it is

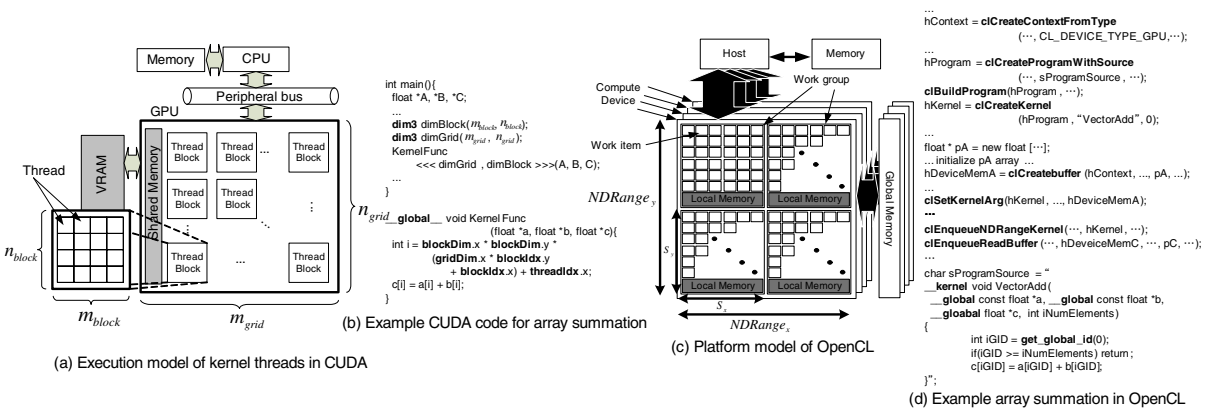


Fig. 2. CUDA and OpenCL architecture models and its vector summation examples.

executed on GPU. In the function, the global variables named `gridDim`, `blockDim`, `blockIdx`, `threadIdx`, implicitly declared by the CUDA runtime, are available to be used to specify the size of the grid and the thread block, the indices of the thread block and of the thread respectively. The function is called by the host CPU program specifying the number of threads with `<<< >>>`.

The OpenCL defines a common platform model that includes the processing element and the memory hierarchy. Fig. 2(c) illustrates the platform model for the processing element. The host CPU is connected to the *OpenCL device*. The OpenCL device consists of the individual processing element called *compute unit*. The compute unit includes one or more *work groups* that include the *work items*. The work item is identified by the unique ID and processes the corresponding result using the related input data associated by the ID. The total number of work items is given by the program using a parameter called *NDRange* that can be defined in from one to three dimensions. The example in the figure includes $NDRange_x \times NDRange_y$ work items. The OpenCL program is written in C as the host CPU side shown in Fig. 2(d). The resources in the OpenCL are obtained by the *context* created by the runtime function. In the figure, the context for a GPU is defined by specifying `CL_DEVICE_TYPE_GPU` as its argument. The argument is variable to select different types of accelerators. The kernel program is provided by a source string defined as an array of char. The string is passed to the runtime functions to compile and prepare the executable code in the accelerator. The buffers for I/O data streams are allocated using the conventional functions such as "new" or "malloc" in the CPU side. The programmer can select if the buffers are accessed directly from the accelerator or if the buffer mirrors are allocated in the accelerator's memory. And then, the argument pointers of the kernel function that point to the actual buffers in the host and/or in the compute device are passed to the accelerator. Finally, the kernel is executed by the `clEnqueueNDRangeKernel` function and the output data streams in the accelerator's memory are copied to the host CPU side.

The assumed architectures in CUDA and OpenCL are very similar. The major difference between those is the kernel compilation mechanism. The CUDA program is compiled whole code including the kernel function by using the *nvcc* compiler. Then the executable for the host CPU downloads the program implicitly to the GPU. Besides, the OpenCL one passes the source string of the kernel code to the runtime function. To separate the kernel code in CUDA as performed by OpenCL, *nvcc* can output the assembly language (PTX) of the kernel code. The assembly language code is also able to be loaded by a runtime function of CUDA. To keep the code compatibility among both runtimes, it is important to develop a unified interface for the accelerator that loads and executes the kernel code without developing the host side program using different runtime functions.

2.3. Caravela Platform

To standardize the programming interfaces of the stream computing, *Caravela platform* was proposed and developed by the author of this paper [9]. It works like a wrapper interface of the stream computing runtimes. Currently, Caravela supports both CUDA and OpenCL for the lower level runtime.

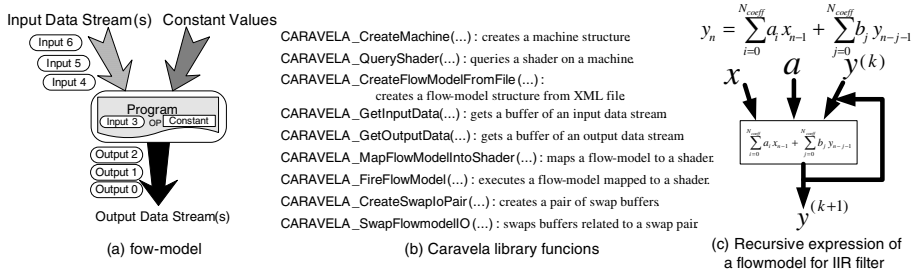


Fig. 3. Caravela platform (a) flow-model that is executed on an accelerator using (b) Caravela library from the host CPU. A recusive algorithm such as (c) IIR filer can be implemented with the swap function in Caravela library defining the *iopair*, .

```
<?xml version="1.0" encoding="ASCII"?>
<FlowModel>
 <Input>
  <Name>a</Name>
  <DataType>INT</DataType>
  <Length>1024</Length>
  <Index>0</Index>
 </Input>
 <Input>
  <Name>b</Name>
  <DataType>INT</DataType>
  <Length>1024</Length>
  <Index>1</Index>
 </Input>
```

Input data stream definition for a and b

```
 <Output>
  <Name>c</Name>
  <DataType>INT</DataType>
  <Length>1024</Length>
  <Index>0</Index>
 </Output>
```

Output data stream definition for c

```
<Kernel>
  __kernel void test(__global int *a, __global int *b, __global int *c) {
    int id = get_global_id(0);
    c[id] = a[id] + b[id];
  }
</Kernel>
```

Kernel function "test" written in OpenCL C.

```
<KernelName>test</KernelName>
```

Specifying the target function name "test"

```
<LangType>SHADERLANG _OPENCL1.0</LangType>
<RuntimeType>RUNTIME_OPENCLGPU</RuntimeType>
```

Specifying the runtime time of the lower layer. Here specifies OpenCL on a GPU.

```
<TotalNumThreads>1024</TotalNumThreads>
<TotalNumBlocks>128</TotalNumBlocks>
```

Specifying the number of thread blocks and the one of threads.
Here specifies 1024 total number of threads in 128 blocks. Each block includes 8 threads.

```
</FlowModel>
```

Fig. 4. An example of flow-model that executes vector summation over OpenCL runtime.

The Caravela platform uses a concept of *flow-model* for programming a given task. Applications are programmed on this platform by using the *Caravela library*, which maps flow-models into the available accelerators. As shown in Fig. 3, the flow-model is composed of 1-dimensional input/output data streams, constant input parameters and a program which processes the input data streams and generates the output data streams. The methods to execute a given task in a flow-model can be encapsulated into an XML file listed in Fig. 3. Properties of the I/O data streams and the kernel program can be specified by tags in an XML file because those are able to be stored in text format. Actually the program is the kernel function written in the language specified by a runtime type tag in the XML executed by the accelerator via the specified lower level runtime.

The Caravela platform is mainly composed by a library that supports an API as shown in Fig. 3(b). The Caravela library has the resource hierarchy definition layered by *Machine* that is a host machine of a peripheral bus adapter, *Adapter* is a peripheral bus adapter that includes one or multiple accelerators and finally *Shader* is an accelerator. The host CPU program of an application needs to map a flow-model into a shader in order to execute it. Fig. 3(b) shows the basic Caravela functions for executing a given flow-model. Using those functions, a programmer can easily implement an application using the framework of flow-models, just by mapping flow-models into shaders.

Besides the basic functions for flow-model execution, the Caravela library also provides data buffering mechanisms implemented within the *buffer swapping functions* [15]. The functions exchange an input data stream with an output data one just by swapping a pair of data structures. This mechanism suits well to recursive applications that feed forward the output result(s) to the input(s) such the IIR (Infinite Impulse Response) filter as shown in Fig. 3(c). The last two functions listed in Fig. 3(b) implement this mechanism by selecting the best method suitable for the stream computing runtime in the lower layer. In the cases of CUDA and OpenCL, it exchanges buffer pointers in the accelerator side without any data copy operation.

Fig. 3 shows an example of flow-model that invokes vector summation over OpenCL runtime with 1024 compute units. Although the kernel program language is different according to the lower level runtime such as CUDA, the execution style in the accelerator is standardized by the flow-model execution framework unified by the Caravela library.

2.4. Discussion

The stream-based computing environment provided by the accelerators brings a powerful parallel processing environment due to the exploited concurrency implicitly from the program. Moreover the runtime environments such as OpenCL and CUDA let the programmer easily develop the application on the accelerators. However, the programmer needs to design and implement both the host CPU program and the kernel program for the accelerator. Caravela platform provides a model-based execution mechanism using flow-model. However, it still has the duty for the double programming separating the CPU program using the Caravela library and the flow-model with the kernel program. So that the programmer concentrates to write the kernel program on the accelerator, it is indispensable to invent a new programming environment where avoids to develop the host CPU program. As the related work, *barracuda* [16] on ruby extension for OpenCL provides wrapper methods for hiding the difficulty of the host CPU program. And StreamIt [17] provides a unified language to resolve the double programming difficulty at compiling time. However those also need to code the scenario for the kernel execution timings. If the programmer needs to invoke multiple kernel programs concurrently, those programs must include a detailed schedule for multi-thread execution on the CPU side. Therefore, it is important to develop a kernel execution environment for intuitive programming of complex applications utilizing accelerator's power, where the programmer only considers the kernel program, the I/O kernels and parallelization parameters. And then the execution timings should be automatically and implicitly decided at running time. In this paper, using the flow-model definition to resolve the problem, we propose a novel shell-like and commandline-based programming environment for the stream-based accelerators called *CarSh*.

3. Commandline execution support for Stream-based Accelerators

3.1. Design of CarSh

When we execute any kind of commands in a conventional CPU-based system, we often use a shell such as *bash*. We just rely on the execution timings and managements of the processes such as background/foreground execution. We consider the same execution mechanism on the stream-based programs for accelerators.

Fig. 5(a) shows the system overview of CarSh. CarSh receives flow-models and the corresponding input data files. After the execution of the flow-model the output data are also stored into files. As shown in Fig. 5(b) CarSh executes flow-models directly (1) from the commandline of CarSh like a shell prompt, (2) from an argument of CarSh and (3) from a batch file of an execution schedule of multiple flow-models. The batch file implements a pipeline execution of multiple flow-models, for example, shown in Fig. 5(c). The data I/O for the pipeline execution are provided by files or internally allocated buffers in CarSh. In the pipeline, *flowmodel2* and *flowmodel3* can be invoked simultaneously. Here, CarSh needs a background concurrent execution mechanism of the flow-models. If a flow-model defines the recursive I/O using the swap pair, CarSh needs automatically to detect the property and executes the iteration for the swap pair. According to the consideration for the interface, we introduce the design considerations below for CarSh.

3.1.1. Management of flow-model execution

CarSh needs to execute a flow-model automatically detecting the definitions and execute it over the Caravela framework. It needs to define an executable format for the execution. The format includes 1) I/O buffer definition with the data type, 2) flow-model XML file, 3) target lower level runtime such as OpenCL, 4) optional functionality definitions such as the *swap function*. CarSh prepares the input data from the I/O definitions, executes the flow-model applying the optional functions and finally saves the result of the output data from the flow-model. This scenario is packed in a single executable format and passed it to CarSh.

In addition to the single execution of a flow-model, CarSh needs a batch execution mode for supporting the pipeline execution of multiple flow-models as shown in Fig. 5(c). The batch execution follows a formatted scenario with the execution steps of the flow-models. To implement this batch execution, CarSh needs a background execution mode and a synchronization function for the previous flow-model executions. For example, to invoke the pipeline in the figure, CarSh executes *flowmodel1* and wait the execution. After that, *flowmodel2* and *flowmodel3* can be concurrently executed as the background tasks. CarSh also needs to have a synchronization function to wait for finishing previously executed flow-models.

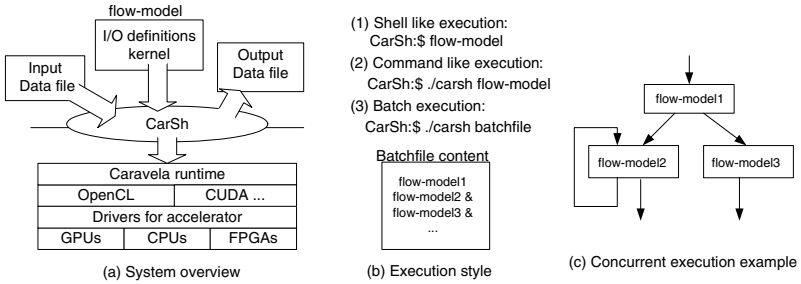


Fig. 5. CarSh system overview.

The flow-model does not define any behavior regarding iterative execution. Moreover, a set of flow-model (i.e. batch) execution must have possibility to be repeated as it would get the results after specified steps such as the LU decomposition algorithm. Therefore, CarSh needs a function to repeat execution of a flow-model or a batch scenario for a specified times.

3.1.2. Management of I/O data streams

The flow-model execution needs input data and after the execution it saves the output data. CarSh introduces two mechanisms for saving/restoring the I/O data. One is a simple mechanism reading/writing the input/output data from a file. In this case, CarSh sets the data read from the file to the corresponding input buffer and uses it for the flow-model execution. The output data are also saved into a file that can be used as the input again. Thus, through one flow-model execution to another in a pipeline structure the files are passed and received. Another mechanism is *virtual buffer* that works as a virtual space for I/O data provided by CarSh inside. The virtual buffer is first prepared by CarSh before the flow-model execution. During the execution, it is used as the place for saving the I/O data of flow-models. The content data of the virtual buffer can be loaded from a file or saved to a file. To manage the virtual buffer, CarSh needs the management functions for *creating*, *deleting* the buffer, and also functions for *filling* and *dumping* data in the buffer from/to files.

According to the functions for flow-model execution and the I/O data, CarSh will provide a shell-like stream computing environment just giving the kernel programs and the execution scenario is packed into the executable or the batch. During execution of the scenario, CarSh fulfills the input data from files or the virtual buffer and passes the execution result to the next file or the virtual buffer. The next flow-model can read the result from the buffer to continue the execution. Thus, the flow-model execution conveys data from one buffer to the next ones. CarSh provides also the iteration of flow-model or batch. Thus, the programmer who uses CarSh does not consider the host CPU program at all. He/she finally becomes available to focus on designing just the kernel programs and the dataflow scenario.

3.2. Implementation of CarSh

Our first implementation of CarSh employs *process*-based task management using *fork* system call on Linux environment. Thread-based implementation is also possible and would achieve better performance. A process is assigned to each flow-model execution by giving an executable XML file as shown in Fig. 6(a) to CarSh commandline. When '&' is added in the last of the commandline, the process is executed in background. This implements the concurrent execution of multiple flow-models. For the synchronization of one or more process executions our implementation introduces *sync* command to wait all the process execution including the background processes using `waitpid` system call. The batch scenario is written in an XML file with `<CarshBat>` tag. Given in the commandline in CarSh, the batch XML file includes the steps of the scenario like the example of Fig. 6(b).

I/O data for the flow-model are loaded and saved from/to CSV files. The file is directly assigned by `<DataFile>` tag in `<Input>` and `<Output>` tags. The `<DataFile>` tag accepts also the virtual buffer name as the input for the flow-model. The I/O arguments of the kernel code in the flow-model are linked in the executable file of CarSh. Thus, the virtual buffers are connected to the I/O in the flow-model. This means that the I/O data inputted/outputted to/from accelerator will be passed to/from the CSV files.

```
<?xml version="1.0" encoding="ASCII"?>
<CarshEx>
 <ModelFile>Flow-model XML file name</ModelFile>
 <Input>
  <Name>Input valuable name in kernel function</Name>
  <DataFile>CSV file name</DataFile>
 </Input>
 <Input>
  ....
 </Input>
 <Output>
  <Name>Output valuable name in kernel function</Name>
  <DataFile>CSV file name</DataFile>
 </Output>
 <SwapPair>
  <Input>Input valuable name in kernel function</Input>
  <Output>Output valuable name in kernel function</Output>
  <NumSwap>Number of swaps</NumSwap>
 </SwapPair>
</CarshEx>
```

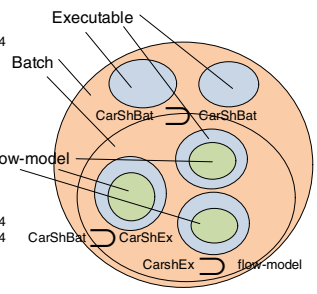
(a) CarSh executable format in XML

```
<?xml version="1.0" encoding="ASCII"?>
<CarshBat>
virtbuf create int 1024 buf_a
virtbuf create int 1024 buf_b
virtbuf fill sample 1_a.csv buf_a
virtbuf fill sample 1_b.csv buf_b
virtbuf create int 1024 buf_c
sample_virt &amp;amp;
sync
repeat 10 sample_virt.xml
sync
virtbuf dump c_tmp.csv buf_c
virtbuf swap buf_a buf_c
virtbuf delete buf_a
virtbuf delete buf_b
virtbuf delete buf_c
exit
</CarshBat>
```

(b) CarSh batch format in XML

```
$ ./carsh
Starting CarSh (Caravela Shell) ...
CarSh: $ virtbuf create int 1024 buf_a
CarSh: $ virtbuf list
1: Name=buf_a, DataType=int, Length=1024
CarSh: $ virtbuf create int 1024 buf_b
CarSh: $ virtbuf fill sample1_a.csv buf_a
CarSh: $ virtbuf fill sample1_b.csv buf_b
CarSh: $ virtbuf create int 1024 buf_c
CarSh: $ sample_virt &
CarSh: $ sync
CarSh: $ repeat 10 sample_virt.xml
CarSh: $ virtbuf dump c_tmp.csv buf_c
CarSh: $ virtbuf delete buf_a
CarSh: $ virtbuf list
1: Name=buf_c, DataType=int, Length=1024
2: Name=buf_b, DataType=int, Length=1024
CarSh: $ virtbuf delete buf_b
CarSh: $ virtbuf delete buf_c
CarSh: $ exit
```

(c) An example of commandline executions



(d) relations among executable, batch and flow-model

Fig. 6. Implementation of CarSh.

The virtual buffer is implemented by the POSIX shared memory object. The first process of CarSh (here calls this process *master*) creates the shared objects that correspond to the virtual buffers using `shm_open` system call and the buffer sizes are truncated by `ftrancate` system call. The master process folks other children processes of flow-models that open the shred memory objects, and then the children processes use the shared memory object as the I/O buffers. The content of the shared object is saved into a file in CSV format. While the flow-model execution processes are working, the virtual buffers are never removed because those are used for the flow-model execution. Then the master process can delete the buffers using `shm_unlink` system call. Thus the shared object is created and deleted by the master process.

Fig. 6(c) shows an example of CarSh commandline execution. First the *virtbuf* command manages the virtual buffers with the argument *list* that shows all allocated virtual buffers and the argument *create* that allocates a virual buffer. After executing an executable or a batch file named *sample_virt.xml* in the background and synchronizes it by *sync* command. *repeat* performs the iterative execution of the same XML file for 10 times. The *virtbuf* command can save the buffer contents to files using *dump* argument. The *swap* argument of the command exchanges buffer names. Finally the *delete* argument deletes all the allocated virtual buffers.

The inclusion relations among the flow-model, the executable and the batch are illustrated in Fig. 6(d). The executable only includes the flow-model directly. The batch includes the executables and the other batches. If a programmer needs to repeat execution of a processing pipeline with flow-models, he/she can pack the pipeline to a batch and then prepares another batch that includes the batch. Therefore, nesting the executable and the batch, we can implement any combination of processing flows.

4. Evaluations

Let us explain evaluations of CarSh focusing on the performance and the programmability. Here we use a typical image filtering using 2D FFT performed often in the image operations. Fig. 7(a) shows the processing steps used in the evaluation. An image (*Lena*), which is transformed by the FFT, is passed to a high or low pass filter, and finally transposed by IFFT. This simple process is composed by five flow-models: *reorder* performs butterfly exchanges, *transpose* inverses the rows and the columns, *filter, FFT and IFFT*. the whole calculation is defined by a CarSh batch XML file listed in Fig. 7(b). It uses virtual buffers for the real and imaginary parts inputted/outputted to/from the subsequent processes managed by *virtbuf* command such as Fig. 7(b)-(1)(2). After every flow-model execution, those buffers are exchanged such as Fig. 7(b)-(3). After the executions of the flow-models, the virtual buffers are deleted. Here, each flow-model execution is called from the CarSh executable XML file, for example, FFT shown in Fig. 7(c). The I/O data for initialization/resulting values are passed via the virtual buffers. The flow-model of the FFT is shown in Fig. 7(d). The I/O arguments of the kernel program correspond to the real and imaginary parts. Those match each other among the executable and the flow-model. Finally, CarSh will execute the batch XML file to get a filtered image.

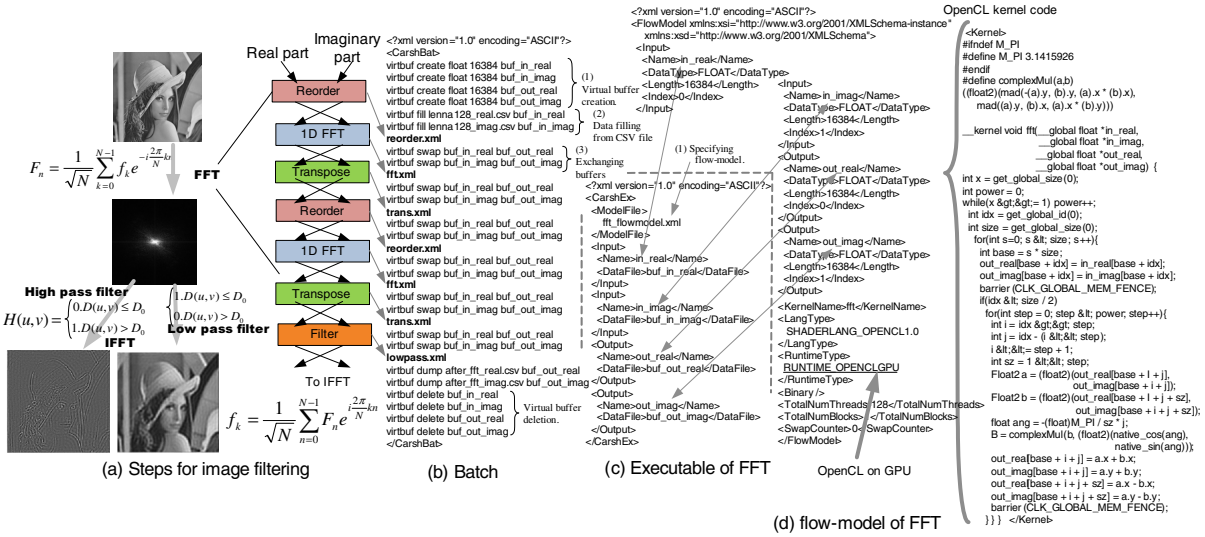


Fig. 7. Implementation of image filtering using FFT on CarSh.

4.1. Programmability of CarSh

This example is a typical pipelined application with multiple flow-models. The pipeline is organized by passing buffers from one kernel program to the next. This must be implemented by buffer management functions on the host CPU side if we apply the conventional double programming method using the OpenCL runtime for accelerators. Moreover, CarSh provides the virtual buffer. It is easy for the programmer to allocate buffers used for the I/O data from/to the flow-model. Thus, buffers are easily defined in the batch XML file by specifying the names. Those are fulfilled from CSV files to the buffer, and simply passed via the names of the buffer among flow-models in the executables.

Without considering the timings for kernel executions and also the buffer management among the host CPU and the accelerator, the programmer can perform a straightforward programming using CarSh framework. The programmer is able to focus on brushing up the stream-based algorithm written in the kernel program. Moreover, because whole code for CarSh is written in text, it is very highly portable among different combinations of a host CPU and an accelerator. This enables a remote development environment where the code compatibility is guaranteed. For example, when we use different kinds of accelerators over OpenCL, we can just change `RuntimeType` tag in flow-model as shown in Fig. 7(d). This mechanism makes the performance check much easy as we just change the string in the flow-model XML description.

4.2. Performance of CarSh

We have measured performances of the image filtering example with varying the accelerator types and the image sizes. Our platform of the performance test is a PC with a Corei7 930 2.80GHz with an Nvidia Tesla M2050 GPU. Table 1 shows the comparison among the GPU-based and the CPU-based executions of the image filtering batch file on CarSh. Both executions use the same kernel functions and the CarSh related XML files. The number of parallelism in OpenCL is set to 1024 where the OpenCL runtime distributes 1024 concurrent threads on the GPU and the CPU. The CPU-based execution is performed on the Intel OpenCL runtime using the multiple cores of the CPU. According to the performances listed in the table, the GPU-based performance achieves almost double of the CPU-based one. This implies that we can control the performance of a set of CarSh executable XMLs. Therefore, if we introduce a new powerful accelerator, we can easily upgrade the performance changing the runtime type description in the flow-model.

As we explained in this section, CarSh brings a simple and transparent programming style for the high programmability on the stream computing employing XML-based packaging for the kernel function invoked in the accelerator. It is easy to control the performance by changing the runtime type description defined in flow-model.

Table 1. Performance of Image filtering using FFT on CarSh.

Image size (OpenCL GPU)	FFT	Filter	IFFT	Total (sec)	Image size (OpenCL CPU)	FFT	Filter	IFFT	Total (sec)
128^2	0.766	0.128	0.764	1.658	128^2	1.969	0.281	1.971	4.221
256^2	0.78	0.128	0.779	1.687	256^2	2.002	0.266	1.999	4.267
512^2	0.825	0.135	0.826	1.786	512^2	2.029	0.302	2.032	4.363
1024^2	0.985	0.153	0.988	2.126	1024^2	2.216	0.329	2.218	4.763

Thus, we have confirmed that CarSh overcomes the programming complexity on the current stream-based accelerators that enforces the double programming. Moreover, CarSh provides the novel commandline interface for executing the kernel function. This promotes high productivity of programs on manycore architectures.

5. Conclusions

To eliminate the programming difficulty on the current manycore accelerators such as GPU and the OpenCL accelerators on FPGA, we have proposed a new commandline support tool, called CarSh. It provides an execution mechanism of the kernel programs on the accelerator just providing XML-based executable files using the flow-model applied from Caravela framework. According to the evaluations using an image filtering application, we confirmed the high programmability and the performance flexibility of CarSh. For the future works, because CarSh executable provides highly compatibility for migrating the application code among different platforms, we are considering to develop a heterogeneous parallel computing environment with CarSh. It will also promote a new secure computing concept because the CarSh executable is best fit to compression and encryption due to the string-based interface. Therefore, it would provide a very secure computing environment that preserves high performance potential.

Acknowledgements

This work is partially supported by the Japan Science Technology Agency (JST) PRESTO program. And also this work is partially supported by KAKENHI (24300020) Grant-in-Aid for Scientific Research (B).

References

[1] D. B. Kirk, W. W. Hwu, Programming Massively Parallel Processors: A Hands-on Approach, Morgan Kaufman, 2010.
[2] N. Hubert, GPU Gems3, 1st Edition, Addison-Wesley Professional, 2007.
[3] J. Gummaraju, M. Rosenbluml, Stream Programming on General-purpose Processors, in: In 38th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), 2005, pp. 343–354.
[4] NVIDIA Corporation, CUDA: Compute Unified Device Architecture programming guide, http://developer.nvidia.com/cuda.
[5] OpenCL, http://www.khronos.org/opencl/.
[6] A. Munshi, B. R. Gaster, T. G. Mattson, J. Fung, D. Ginsburg, OpenCL Programming Guide, Addison Wesley, 2011.
[7] S. Yamagiwa, Invitation to a Standard Programming Interface for Massively Parallel Computing Environment: OpenCL, International Journal of Networking and Computing 2 (2) (2012) 188–205.
[8] S. Yamagiwa, L. Sousa, Design and Implementation of a Stream-based DistributedComputing Platform using Graphics Processing Units, in: ACM International Conference on Computing Frontiers, 2007.
[9] S. Yamagiwa, L. Sousa, Caravela: A Novel Stream-Based Distributed Computing Environment, IEEE Computer 40 (5) (2007) 70–77.
[10] P. Pacheco, An Introduction to Parallel Programming, Morgan Kaufman, 2011.
[11] OpenGL Architecture Review Board, D. Shreiner, M. Woo, J. Neider, T. Davis, OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 3 and 3.1, Addison Wesley, 2009.
[12] D. Wolff, OpenGL 4.0 Shading Language Cookbook : Over 60 Highly Focused, Practical Recipes to Maximize Your Use of the OpenGL Shading Language, Packt open source, Packt Publishing, 2011.
[13] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Kruger, A. E. Lefohn, T. J. Purcell, A survey of general-purpose computation on graphics hardware, in: Eurographics 2005, State of the Art Reports, 2005, pp. 21–51.
[14] Altera, OpenCL for Altera FPGAs: Accelerating Performance and Design Productivity, http://www.altera.com/products/software/opencl/opencl-index.html.
[15] S. Yamagiwa, L. Sousa, D. Antao, Data Buffering Optimization Methods toward a Uniform Programming Interface for GPU-based Applications, in: CF '07: Proceedings of the 4th International Conference on Computing Frontiers, ACM Press, 2007, pp. 205–212.
[16] Barracuda: An OpenCL Library for Ruby, http://gnuu.org/2009/08/30/barracuda-an-opencl-library-for-ruby/.
[17] W. Thies, M. Karczmarek, S. P. Amarasinghe, StreamIt: A Language for Streaming Applications, in: Proceedings of the 11th International Conference on Compiler Construction, Springer-Verlag, 2002, pp. 179–196.